Parallel Merge Sort

by

Richard Cole

Ultracomputer Note #115
Computer Science Technical Report #278
March, 1987

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012

Parallel Merge Sort

by

Richard Cole

Ultracomputer Note #115
Computer Science Technical Report #278
March, 1987

This work was supported in part by an IBM Faculty Development Award and by NSF grant DCR-84-01633.

ABSTRACT

We give a parallel implementation of merge sort on a CREW PRAM that uses n processors and O(logn) time; the constant in the running time is small. We also give a more complex version of the algorithm for the EREW PRAM; it also uses n processors and O(logn) time. The constant in the running time is still moderate, though not as small.

1. Introduction

There are a variety of models in which parallel algorithms can be designed. For sorting, two models are usually considered: circuits and the PRAM; the circuit model is the more restrictive. An early result in this area was the sorting circuit due to Batcher [B-68]; it uses time 1/2 $\log^2 n$. More recently, [AKS-83] gave a sorting circuit that ran in O(logn) time; however, the constant in the running time was very large (we will refer to this as the AKS network). The huge size of the constant is due, in part, to the use of expander graphs in the circuit. The recent result of Lubotsky et al [LPS-86] concerning expander graphs may well reduce this constant considerably; however, it appears that the constant is still large [CO-86, Pa-87].

The PRAM provides an alternative, and less restrictive, computation model. There are three variants of this model that are frequently used: the CRCW PRAM, the CREW PRAM, and the EREW PRAM; the first model allows concurrent access to a memory location both for reading and writing, the second model allows concurrent access only for reading, while the third model does not allow concurrent access to a memory location. A sorting circuit can be implemented in any of these models (without loss of efficiency).

Preparata [Pr-78] gave a sorting algorithm for the CREW PRAM that ran in O(logn) time on (n logn) processors; the constant in the running time was small. (In fact, there were some implementation details left incomplete in this algorithm; this was rectified by Borodin and Hopcroft in [BH-82].) Preparata's algorithm was based on a merging procedure given by Valiant [V-75]; this procedure merges two sorted arrays, each of length at most n, in time O(loglogn) using a linear number of processors. When used in the obvious way, Valiant's procedure leads to an implementation of merge sort on n processors using O(logn loglogn) time. More recently, Kruskal [K-83] improved this sorting algorithm to obtain a sorting algorithm that ran in time O(logn loglogn/logloglogn) on n processors. (The basic algorithm was Preparata's; however, a different choice of parameters was made.) In part, Kruskal's algorithm depended on using the most efficient versions of Valiant's merging algorithm; these are also described in Kruskal's paper. More recently, Bilardi and Nicolau [BN-86] gave an implementation of bitonic sort on the EREW PRAM that used n/logn processors and $O(\log^2 n)$ time. The constant in the running time was small.

In the next Section, we describe a simple CREW PRAM sorting algorithm that uses n processors and runs in time O(logn); the algorithm performs just 5/2nlogn comparisons. In Section 3 we modify the algorithm to run on the EREW PRAM. The algorithm still runs in time O(logn) on n processors; however, the constant in the running time is somewhat less small than for the CREW algorithm. We note that apart from the AKS sorting network, the known deterministic EREW sorting algorithms that use about n processors all run in time $O(\log^2 n)$ (these algorithms are implementations of the various sorting networks such as Batcher's sort). Our algorithms will not make use of expander graphs or any related constructs; this avoids the huge constants in the running time associated with the AKS construction.

The contribution of this work is twofold: first, it provides a second O(logn) time, n processor parallel sorting algorithm (the first such algorithm is implied by the AKS sorting circuit); second, it considerably reduces the constant in the running time (by comparison with the AKS result). Of course, AKS is a sorting circuit; this work does not provide a sorting circuit.

In Section 4, we show how to modify the CREW algorithm to obtain CRCW sorting algorithms that run in sublogarithmic time. We will also mention some open problems concerning sorting on the PRAM model in sublogarithmic time. In Section 5, we consider a parametric search technique due to Megiddo [M-83]; we show that the partial improvement of this technique in [C-84] is enhanced by using the EREW sorting algorithm.

2. The CREW Algorithm

By way of motivation, let us consider the natural tree-based merge sort. Consider an algorithm for sorting n numbers. For simplicity, suppose that n is a power of 2, and all the items are distinct. The algorithm will use an n-leaf complete binary tree. Initially, the inputs are distributed one per leaf. The task, at each internal node u of the tree, is to compute the sorted order for the items initially at the leaves of the subtree rooted at u. The computation proceeds up the tree, level by level, from the leaves to the root, as follows. At each node we compute the merge of the sorted sets computed at its children. Using the O(loglogn) time, n processor merging algorithm of Valiant, will yield an O(lognloglogn) time, n processor sorting algorithm. In fact, we know there is an Ω(loglogn) time lower bound for merging two sorted arrays of n items using n processors [BH-82], thus we do not expect this approach to lead to an O(logn) time, n processor sorting algorithm.

We will not use the fast O(loglogn) time merging procedure; instead, we base our algorithm on an O(logn) time merging procedure, similar to the one described in the next few sentences. The problem is to merge two sorted arrays of n items. We proceed in logn stages. In the ith stage, for each array, we take a sorted sample of 2^{i-1} items, comprising every $n/2^{i-1}$th item in the array. We compute the merge of these two samples. Given the results of the merge from the $i-1$st stage, the merge in the ith stage can be computed in constant time (this, or rather a related result, will be justified later).

At present, this merging procedure merely leads to an $O(\log^2 n)$ time sorting algorithm. To obtain an O(logn) time sorting algorithm we need the following key observation:

The merges at the different levels of the tree can be pipelined.

This is plausible because merged samples from one level of the tree provide fairly good samples at the next level of the tree. Making this statement precise is the key to the CREW algorithm.

We now describe our sorting algorithm. The inputs are placed at the leaves of the tree. Let u be an internal node of the tree. The task, at node u, is to compute L(u), the sorted array that contains the items initially at the leaves of the subtree rooted at u. At intermediate steps in the computation, at node u, we will have computed UP(u), a sorted subset of the items in L(u); UP(u) will be stored in an array also. The items in UP(u) will be a rough sample of the items in L(u). As the algorithm proceeds, the size of UP(u) increases, and UP(u) becomes a more accurate approximation of L(u). (Note that at each stage we use a different array for UP(u).)

We explain the processing performed in one stage at an arbitrary internal node u of the tree. The array UP(u) is the array at hand at the start of the stage; NEWUP(u) is the array at hand at the start of the next stage, and OLDUP(u) is the array at hand at the start of the previous stage, if any. Also, in each stage, we will create an array SUP(u) (short for SAMPLEUP(u)) at node u; NEWSUP(u), OLDSUP(u) are the corresponding arrays in respectively, the next, and previous, stage. At each stage, for each node u, the computation comprises the following two phases.

(1) Form the array SUP(u); it comprises every fourth item in UP(u), in sorted order.

(2) Let v and w be u's children. Compute NEWUP(u) = SUP(v) $\cup$ SUP(w), where $\cup$ denotes merging.

There are some boundary cases where we need to change Phase 1. (For example, initially, the UP arrays each contain one or zero items. Thus the SUP arrays would all be empty and the algorithm would do nothing.) In view of this, we establish the following goal: at each stage, so long as $0 \neq |UP(u)| \neq |L(u)|$, the size of NEWUP(u) will be twice the size of UP(u). At this point, some definitions will be helpful. A node is *external* if $|UP(u)| = |L(u)|$, and it is *inside* otherwise. Phases 1 and 2, above, are performed at each inside node. At external nodes, Phase 2 is not performed and Phase 1 is modified as follows. For the first stage in which u is external, Phase 1 is unchanged. For the second stage, SUP(u) is defined to be every second item in UP(u), in sorted order. And for the third stage, SUP(u) is defined to be every item in UP(u), in sorted order. It should be clear that we have achieved our goal, namely:

Lemma 1: While $0 < |UP(u)| < |L(u)|$, $|NEWUP(u)| = 2|UP(u)|$.

It is also clear that 3 stages after node u becomes external, node t, the parent of u, also becomes external. We conclude:

Lemma 2: The algorithm has 3logn stages.

It remains for us to show how to perform the merges needed for Phase 2. We will show that they can be performed in constant time using O(n) processors. This yields the O(logn) running time for the sorting algorithm.

A few definitions will be helpful. Let e, f, g be three items, with $e<g$. f is *between* e and g if $e \leq f$ and $f<g$; we also say that e and g *straddle* f. Let L and J be sorted arrays. Let f be an item in J, and let e and g be the two adjacent items in L that straddle f (if necessary, we let $e=-\infty$ or $g=\infty$); then the *rank* of f in L is defined to be the rank of e in L (if $e=-\infty$, f is defined to have rank 0). We define the range (e,g] to be the interval

induced by item g (including the cases $e = -\infty$ and $g = \infty$). L is a c-*cover* of J if each interval induced by an item in L contains at most c items from J. We also say that the items from J in the range (e,g] are *contained in* g's interval. We define L and J to be *cross-ranked* if for each item in J (respectively L) we know its rank in L (respectively J).

We will need the following observation to show that the merge can be performed in O(1) time:

> OLDSUP(v) is a 3-cover for SUP(v) for each node v; hence UP(u) is a 3-cover for each of SUP(v) and SUP(w), since UP(u) = OLDSUP(v) ∪ OLDSUP(w)).

This will be shown in Corollary 1, below. But first, we describe the merge (Phase 2).

We need some additional information in order to perform the merge quickly. Specifically, for each inside node u, for each item from UP(u), we provide its rank in each of SUP(v) and SUP(w). Using these ranks, first we show how to compute NEWUP(u), and second we show, for each item from NEWUP(u), how to compute its ranks in NEWSUP(v) and NEWSUP(w).

Step 1 -- Computing NEWUP(u). Let e be an item in SUP(v); the rank of e in NEWUP(u) = SUP(v) ∪ SUP(w) is equal to the sum of its ranks in SUP(v) and SUP(w). So to compute the merge we cross-rank SUP(v) and SUP(w) (the method is given in the following two paragraphs). At this point, for each item e in SUP(v), besides knowing its rank in NEWUP(u), we know the two items d and f in SUP(w) that straddle e, and we know the ranks of d and f in NEWUP(u) (these will be needed in Step 2). For each item in NEWUP(u) we record whether it came from SUP(v) or SUP(w) and we record the ranks (in NEWUP(u)) of the two straddling items from the other set.

Let e be an item in SUP(v); we show how to compute its rank in SUP(w). We proceed in two substeps. First, for each item in SUP(v) we compute its rank in UP(u). This task is performed by processors associated with the items in UP(u), as follows. Let y be an item in UP(u). Consider the interval I(y) in UP(u) induced by y, and consider the items in SUP(v) contained in I(y) (there are at most 3 such items by the 3-cover property). Each of these items is given its rank in UP(u) by the processor associated with y. This substep takes constant time if we associate one processor with each item in the UP array at each inside node.

In the second substep we compute the rank of e in SUP(w). We determine the two items d and f in UP(u) that straddle e, using the rank computed in the first substep. Suppose that d and f have ranks r and t, respectively, in SUP(w). Then all items of rank r or less are smaller than item e (recall we assumed that all the inputs were distinct), while all items of rank greater than t are larger than item e; thus the only items about which there is any doubt as to their size relative to e are the items with rank s, $r < s \leq t$. But there are at most 3 such items by the 3-cover property. By means of at most 2 comparisons, the relative order of e and these (at most) 3 items can be determined. So the second substep requires constant time if we associate one processor with each item in each SUP array.

Step 2 -- maintaining ranks. For each item e in NEWUP(u), we want to determine its rank in NEWSUP(v) We start by making a few observations. Given the ranks for an item from UP(u) in both SUP(v) and SUP(w) we can immediately deduce the rank of this item in NEWUP(u) = SUP(v) ∪ SUP(w) (the new rank is just the sum of the two old ranks). Similarly, we obtain the ranks for items from UP(v) in NEWUP(v). This yields the ranks of items SUP(v) in NEWSUP(v) (for each item in SUP(v) came from UP(v), and NEWSUP(v) comprises every fourth item in NEWUP(v)). Thus for every item e in NEWUP(u) that came from SUP(v) we have its rank in NEWSUP(v); it remains to compute this rank for those items e in NEWUP(u) that came from SUP(w).

Recall that for each item e from SUP(w) we computed the straddling items d and f from SUP(v) (in Step 1). We know the ranks r and t of d and f, respectively, in NEWSUP(v) (as asserted in the previous paragraph). Every item of rank r or less in NEWSUP(v) is smaller than e, while every item of rank greater than t is larger than e; thus the only items about which there is any doubt concerning their size relative to e are the items with rank s, $r < s \leq t$. But there are at most 3 such items by the 3-cover property. As before, the relative order of e and these (at most) 3 items can be determined by means of at most 2 comparisons. Thus Step 2 takes constant time if we associate a processor with each item in each NEWUP array.

It remains to prove the 3-cover property (Corollary 1 to Lemma 3) and to determine the complexity of the algorithm (Lemmas 4 and 5).

Lemma 3: Let $k \geq 1$. At the start of a stage, any k adjacent intervals in SUP(u) contain at most $2k+1$ items from NEWSUP(u).

Proof: We prove the result by induction on the (implicit) stage number. The claim is true initially, for when SUP(u) first becomes non-empty, it contains one item and NEWSUP(u) contains 2 items, and when SUP(u) is empty, NEWSUP(u) contains at most 1 item.

Inductive step. We seek to prove that k adjacent intervals in NEWSUP(u) contain at most $2k+1$ items from SUP(u), assuming that the result is true for the previous stage, i.e. that for all nodes u′, for all $k' \geq 1$, k′ intervals in OLDSUP(u′) contain at most $2k'+1$ items from SUP(u′).

We first suppose that u is not external at the start of the current stage. Consider a sequence of k adjacent intervals in SUP(u); they cover the same range as some sequence of 4k adjacent intervals in UP(u). Recall that UP(u) = OLDSUP(v) ∪ OLDSUP(w). The 4k intervals in UP(u) overlap some $h \geq 1$ adjacent intervals in OLDSUP(v) and some $j \geq 1$ adjacent intervals in OLDSUP(w), with $h+j=4k+1$. The h intervals in OLDSUP(v) contain at most $2h+1$ items from SUP(v), by the inductive hypothesis, and likewise, the j intervals in OLDSUP(w) contain at most $2j+1$ items from SUP(w). Recall that NEWUP(u) = SUP(v) ∪ SUP(w). Thus the 4k intervals in UP(u) contain at most $2h+2j+2=8k+4$ items from NEWUP(u). But NEWSUP(u) comprises every fourth item in NEWUP(u); thus the k adjacent intervals in SUP(u) contain at most $2k+1$ items from NEWSUP(u).

It remains to prove the Lemma for the first and second stages in which u is external (there is no NEWUP(u) array, and hence no NEWSUP(u) array for the third stage in which u is external). Here we can make the following stronger claim concerning the relationship between SUP(u) and NEWSUP(u): k adjacent intervals in SUP(u) contain at most 2k items from NEWSUP(u) and every item in SUP(u) occurs in NEWSUP(u). This is readily seen. Consider the first stage in which u is external. SUP(u) comprises every fourth item in UP(u) = L(u) and NEWSUP(u) comprises every second item in UP(u). Clearly the claim is true for this stage; the argument is similar for the second stage. □

Taking $k=1$ we obtain:

Corollary 1: SUP(u) is a 3-cover of NEWSUP(u).

Remark: An attempt to prove Lemma 3, in the same way, with a sampling strategy of every second (rather than every fourth) item will not succeed. This explains why we chose the present sampling strategy. It is not the only strategy that will work (another possibility is to sample every eighth item, or even to use a mixed strategy, such as using samples comprising every second and every fourth item, respectively, at alternate levels of the tree); however, the present strategy appears to yield the best constants.

We turn to the analysis of the algorithm. We start by computing the total number of items in the UP arrays. If $|UP(u)| \neq 0$ and v is not external, then $2|UP(u)| = |NEWUP(u)| = |SUP(v)| + |SUP(w)| = 1/4\,(|UP(v)| + |UP(w)|) = 1/2\,|UP(v)|$; that is

Observation: $|UP(u)| = 1/4\,|UP(v)|$. So the total size of the UP arrays at u's level is 1/8 of the size of the UP arrays at v's level, if v is not external.

The observation need not be true at external nodes v. It is true for the first stage in which v is external; but for the second stage, $|UP(u)| = 1/2\,|UP(v)|$, and so the total size of the UP arrays at u's level is 1/4 of the total size of the arrays at v's level; likewise, for the third stage, $|UP(u)| = |UP(v)|$, and so the total size of the UP arrays at u's level is 1/2 of the total size of the UP arrays at v's level.

Thus, on the first stage in which v is external, the total size of the UP arrays is $n + n/8 + n/64 + \cdots = n + n/7$; on the second stage, it is $n + n/4 + n/32 + \cdots = n + 2n/7$; on the third stage, it is $n + n/2 + n/16 + \cdots = n + 4n/7$. Similarly, on the first stage, the total size of the SUP arrays (and hence of the NEWUP arrays) is $n/4 + n/32 + n/256 + \cdots = 2n/7$; on the second stage, it is $n/2 + n/16 + n/128 + \cdots = 4n/7$; on the third stage, it is $n + n/8 + n/64 + \cdots = 8n/7$.

We conclude that the algorithm needs $O(n)$ processors (so as to have a processor standing by each item in the UP, SUP, and NEWUP arrays) and takes constant time. Let us count precisely how many comparisons the algorithm performs.

Lemma 4: The algorithm performs $15/4\, n \log n$ comparisons.

Proof: Comparisons are performed in the second substep of Step 1 and in Step 2. In Step 1, at most 2 comparisons are performed for each item in each SUP array. Over a sequence of three successive stages this is $2 \cdot (2n/7 + 4n/7 + 8n/7) = 4n$ comparisons. In Step 2, at most 2 comparisons are performed for each item in each NEWUP list. Over a sequence of three successive stages this is also 4n comparisons. However, we have overcounted here; on the third stage, in which node u becomes external, we do not perform any comparisons for items in NEWUP(u); this reduces the cost of Step 2 to 2n comparisons.

So we have a total of at most 6n comparisons for any three successive stages. However, we are still overcounting; we have not used the fact that during the second and third stages in which node u is external, SUP(u) is a 2-cover of NEWSUP(u) and every item in SUP(u) occurs in NEWSUP(u) (see the proof of Lemma 3). This implies that in Step 1, for each item in array SUP(u), in the second or third stage in which u is external, at most one comparison need be made (and not two). This reduces the number of comparisons in Step 1, over a sequence of three stages, by $n/2+n = 3/2n$. Likewise, in Step 2, for each item in array NEWUP(u), in the first or second stages in which the children of u are external nodes, at most one comparison is performed. This reduces the number of comparisons in Step 2, over a sequence of three stages, by $n/4+n/2 = 3/4n$. Thus the total number of comparisons, over the course of three successive stages, is $5/2n$ for Step 1, and $5/4n$ for Step 2, a total of $15/4n$ comparisons. □

In order to reduce the number of comparisons to $5/2n\log n$ we need to modify the algorithm slightly. In Step 1, when computing the rank of each item from SUP(v) (resp. SUP(w)) in SUP(w) (resp. SUP(v)), we will allow only the items in SUP(v) to perform comparisons (or rather, processors associated with these items). We compute the ranks for items from SUP(v) as before. To obtain the ranks for items from SUP(w) we need to change both steps. We change Step 1 as follows. For each item h in SUP(w), we compute its rank r in UP(u) as before (the first substep). Let k be the item of rank r in UP(u), and let s be the rank of k in SUP(v). We also store the rank s with item h. s is a good estimate of the rank of h in SUP(v); it is at most 3 smaller than the actual rank. We change Step 2 as follows. Item e in SUP(v), of rank t, communicates its rank to the following, at most three, *receiving items* in SUP(w): those items with rank t in SUP(v) which at present store a smaller estimate for this rank. (These items are determined as follows. Let d and f be the items from UP(u) that straddle e. Let g be the successor of e in SUP(v). The receiving items are exactly those items straddled both by e and g, and by d and f; the second constraint implies that there are at most 3 receiving items for e, by the 3-cover property.) For those items h in SUP(w) that do not receive a new rank from an item in SUP(v), the rank s computed in the modified Step 1 is the correct rank.

This new procedure reduces the number of comparisons in Step 1 by a factor of 2, and leaves unaltered the number of comparisons in Step 2. We conclude:

Lemma 5: The algorithm performs $5/2 n \log n$ comparisons.

We have shown

Theorem 1: There is an CREW PRAM sorting algorithm that runs in $O(\log n)$ time on n processors, performing at most $5/2 n \log n$ comparisons.

Remark: The algorithm needs only $O(n)$ space. For although each stage requires $O(n)$ space, the space can be reused from stage to stage.

3. The EREW Algorithm

The algorithm from Section 2 is not EREW at present. While it is possible to modify Step 1 of a phase so that it runs in constant time on an EREW PRAM, the same does not appear to be true for Step 2. Since we use a somewhat different merging procedure here, we will not explain how Step 1 can be modified. However, we do explain the difficulty faced in making Step 2 run in constant time on an EREW PRAM. The explanation follows. Consider NEWUP(u) and consider e and g, two items adjacent in SUP(v); suppose that in NEWUP(u), between e and g, there are many items f from SUP(w). Let f′ be an item in NEWSUP(v), between e and g. The difficulty is that for each item f we have to decide the relative order of f and f′; furthermore, the decision must be made in constant time, without read conflicts, for every such item f. This cannot be done. To obtain an optimal logarithmic time EREW sorting algorithm we need to modify our approach. Essentially, the modification causes this difficult computation to become easy by precomputing most of the result.

We now describe the EREW algorithm precisely. We use the same tree as for the CREW algorithm. At each node v of the tree we maintain 2 arrays: UP(v) (defined as before), and DOWN(v) (to be defined). We define the array SUP(v) as before; we introduce a second sample array, SDOWN(v): it comprises every fourth item in DOWN(v). Let u, w, x and y, be respectively, the parent, sibling, left child, and right child of v. A stage of the algorithm comprises the following three steps, performed at each node v.

(1) Form the arrays SUP(v) and SDOWN(v).

(2) Compute NEWUP(v) = SUP(x) $\cup$ SUP(y).

(3) Compute NEWDOWN(v) = SUP(w) $\cup$ SDOWN(u).

We will need to maintain some other arrays in order to perform the merges in constant time; namely, the arrays UP(v) ∪ SDOWN(v) and SUP(v) ∪ DOWN(v). It is useful to note that SDOWN(v) is a 3-cover of NEWSDOWN(v); the proof of this result is identical to the proof of the 3-cover property for the SUP arrays (given in Lemma 3 and Corollary 1).

We describe the new merging procedure. First, we need to introduce some notation. We write J × K to denote that the arrays J and K are cross-ranked. We show how to compute J × K in constant time using a linear number of processors (this yields L = J ∪ K), supposing that we are given the following arrays and cross-ranks:

(i) Arrays SJ and SK that are 3-covers for J and K, respectively. Also, for each item in SJ (resp. SK) we have its rank in J (resp. K).

(ii) J × SK and SJ × K.

(iii) SJ × SK and SL = SJ ∪ SK.

The interval between two adjacent items, e and f, from SL = SJ ∪ SK contains at most 3 items from each of J and K. In order to cross-rank J and K, it suffices, for each such interval, to determine the relative order of the (at most) 6 items it contains. Such an interval, without loss of generality, can be assumed to have one of the following two forms:

(a) Both endpoints are in SJ.

(b) The left endpoint is in SJ and the right endpoint is in SK.

We show how to merge the (at most) 6 items contained in the interval in each of these cases. There are two substeps: in Substep 1 we identify the two sets of (at most) 3 items to be merged, and in Substep 2 we perform the merge.

Substep 1.

Case (a). The (at most) 3 items from J are those straddled by e and f (they are readily determined from (i)). The (at most) 3 items from K, again, are those straddled by e and f (they are readily determined using SJ × K from (ii)).

Case (b). We obtain the (at most) 3 items from J by examining J × SK: we use the pointer from the left (SJ) endpoint into J to provide the left endpoint in J × SK, and we use the right (SK) endpoint to give the right endpoint in J × SK. All the items in J ∪ SK between these two endpoints are items in J. The (at most) 3 items from K are obtained similarly, by examining SJ × K.

Substep 2. It remains to perform a set of merge operations, the output of each merge comprising at most 6 items. Each merge requires at most 5 comparisons, and a constant number of other operations.

To carry out this procedure we associate one processor with each interval in the array SL. The number of intervals is one larger than the number of items in this array.

Remark: If SJ (resp. SK) is a 4-cover of J (resp. K) but SJ (resp. SK) is contained in J (resp. K) then the same algorithm can be used, since the interior of any interval in SL will still contain at most 3 items from J and at most 3 items from K.

We suppose that the following cross-ranks are available at the start of a phase at each node v:

SUP(v) × SDOWN(v), SUP(v) × DOWN(v), UP(v) × SDOWN(v),
OLDSUP(v) × OLDSUP(w), OLDSUP(w) × OLDSDOWN(u).

In addition, we note that since DOWN(v) = OLDSUP(w) ∪ OLDSDOWN(u), and as we have SUP(v) × DOWN(v) and OLDSUP(w) × OLDSDOWN(u), we can immediately obtain:

SUP(v) × OLDSUP(w), SUP(v) × OLDSDOWN(u).

Likewise, from UP(u) = OLDSUP(v) ∪ OLDSUP(w), UP(u) × SDOWN(u) (at node u), and OLDSUP(v) × OLDSUP(w), we obtain

OLDSUP(w) × SDOWN(u).

The computation proceeds in 5 steps.

Step 1. Compute SUP(v) × SUP(w) (yielding NEWUP(u)). We already have:

(i) OLDSUP(v) × OLDSUP(w).

(ii) SUP(v) × OLDSUP(w).

(iii) OLDSUP(v) × SUP(w) (from node w).

Step 2. Compute SUP(w) × SDOWN(u) (yielding NEWDOWN(v)). We already have:

(i) OLDSUP(w) × OLDSDOWN(u).

(ii) OLDSUP(w) × SDOWN(u).

(iii) SUP(w) × OLDSDOWN(u) (from node w).

Step 3. Compute NEWSUP(v) × NEWSDOWN(v). We already have:

(i) SUP(v) × SDOWN(v).

(ii) SUP(v) × NEWDOWN(v), and hence SUP(v) × NEWSDOWN(v)
(this is obtained from: SUP(v) × SUP(w), from Step 1, and SUP(v) × SDOWN(u), from Step 2 at node w, yielding SUP(v) × [SUP(w) ∪ SDOWN(u)] = SUP(v) × NEWDOWN(v)).

(iii) NEWUP(v) × SDOWN(v), and hence NEWSUP(v) × SDOWN(v)
(this is obtained from SUP(x) × SDOWN(v), from Step 2 at node y, and SUP(y) × SDOWN(v), from Step 2 at node x, yielding [SUP(x) ∪ SUP(y)] × SDOWN(v) = NEWUP(v) × SDOWN(v).

Step 4. Compute NEWUP(v) × NEWSDOWN(v). We already have:

(i) NEWSUP(v) × SDOWN(v) (from Step 3(iii)).

(ii) NEWUP(v) × SDOWN(v) (from Step 3(iii)).

(iii) NEWSUP(v) × NEWSDOWN(v) (from Step 3).

(Here NEWSUP(v) is a 4-cover of NEWUP(v), contained in NEWUP(v); as explained in the remark following the merging procedure, this leaves the complexity of the merging procedure unchanged.)

Step 5. Compute NEWSUP(v) × NEWDOWN(v). We already have:

(i) SUP(v) × NEWSDOWN(v) (from Step 3(ii)).

(ii) SUP(v) × NEWDOWN(v) (from Step 3(ii)).

(iii) NEWSUP(v) × NEWSDOWN(v) (from Step 3).

We conclude that each stage can be performed in constant time, given one processor for each item in each array named in (i) of each step.

It remains to show that only O(n) processors are needed by the algorithm. This is a consequence of the following linear bound on the total size of the DOWN arrays.

Lemma 6: $|\text{DOWN}(v)| \le 16/31\,|\text{SUP}(v)|$.

Proof: This is readily verified by induction on the stage number. □

Corollary 2: The total size of the DOWN arrays, at any one stage, is $O(n)$.

This algorithm, like the CREW algorithm, has $3\log n$ stages. We conclude

Theorem 2: There is an EREW PRAM algorithm for sorting n items; it uses n processors and $O(n)$ space, and runs in $O(\log n)$ time.

We do not give the elaborations to the merging procedure required to obtain a small total number of comparisons (as regards constants). We simply remark that it is not difficult to reduce the total number of comparisons to less than $5n\log n$ comparisons; however, as there is no corresponding reduction in the number of other operations, this complexity bound gives a misleading impression of efficiency (which is why we do not strive to attain it).

4. A Sublogarithmic Time CRCW Algorithm

[AAV-86] give tight upper and lower bounds for sorting in the parallel comparison model; using $p \geq n$ processors to sort n items the bound is $\Theta(\frac{\log n}{\log 2p/n})$. In addition, there is a lower bound of $\Omega(\frac{\log n}{\log\log n})$ for sorting n items with a polynomial number of processors in the CRCW PRAM model [BH-87]. We give a CRCW algorithm that uses time $O(\frac{\log n}{1+\log\log 2p/n})$ for $n \leq p \leq n^2$. It is not clear which, if any, of the upper and lower bounds are tight for the CRCW PRAM model.

We describe the algorithm; it is very similar to the CREW algorithm. Let $r = p/n$. It is convenient to assume that n is a power of r (the details of the general case are left to the reader). There are three major changes to the CREW algorithm. First, rather than use a binary tree to guide the merges, we use an r-way tree. This tree has height $h = \log n/\log r$. Second, we define the array SUP(v) to comprise every r^2 item in UP(v), rather than every fourth item (with the natural modification at the external nodes). Third, the array NEWUP(v) is defined to be the r-way merge of the SUP arrays at its r children. As before, the algorithm will have $3h = 3\log n/\log r$ stages. But here, rather than use constant time, each stage will take $O(\log r/\log\log 2r)$ time.

To obtain the r-way merge we perform $1/2r(r-1)$ pairwise merges of the r SUP arrays. To obtain the rank of an item in the array NEWUP(v) we sum its ranks in each of the r SUP arrays. We use r processors to compute this sum; they take O(logr/loglogr) time on a CRCW PRAM using the summation algorithm from [CV-87].

Before explaining how to perform a single pairwise merge it is useful to prove a cover property.

Lemma 7: k intervals in SUP(v) contain at most rk + 1 items in NEWSUP(v).

Proof: The proof is very similar to that of Lemma 2; the details are left to the reader. □

Corollary 3: SUP(v) is an (r+1)-cover of NEWSUP(v).

We perform the merges essentially as in the CREW algorithm. Here, at the start of a stage, we need to assume that for each node u, for each item in UP(u) we have its rank in the SUP arrays at all r of u's children. Let v, w be children of u. We proceed in two steps, as in the CREW algorithm.

In Step 1, we start be dividing each pairwise merge into a collection of merging subproblems, each of size at most 2(r+1); each subproblem is then solved using Valiant's merging algorithm [Va-75]. To divide a merge into subproblems, we exploit the fact that UP(u) is an (r+1)-cover of SUP(v), as follows. For each child v of u, we label each item in SUP(v) with its rank in UP(u) (this is carried out in constant time by providing each item in UP(u) with r(r+1) processors). A subproblem is defined by those items labeled with the same rank; it comprises two subarrays each containing at most r+1 items. For the problem of merging SUP(v) and SUP(w), for any item e in SUP(v) (resp. SUP(w)) the boundaries of its subproblem can be found as follows: let f and g be the items in UP(u) straddling e (obtained using the rank of e in UP(u)); the ranks of f and g in SUP(v), SUP(w) yield the boundaries of the subproblem. To ensure that the merges performed using Valiant's algorithm each take O(loglog2r) time we need to provide a linear number of processors for each merge. Since each item in SUP(v) participates in exactly r−1 merges it suffices to allocate r−1 processors to each item in each SUP array.

In Step 2 (ranking the items from NEWUP(u) − SUP(v) in NEWSUP(v) for each child v of u) we proceed as follows. Consider an interval I induced by an item e from SUP(v). For each such interval I, we divide each collection of items from NEWUP(u) − SUP(v), contained in I, into sets of r contiguous items, with possibly one smaller set per interval. Using Valiant's algorithm, we merge each such set with the at most r+1 items in NEWSUP(v) contained in I. If we allocate r processors to each merging problem they will take $O(\log\log 2r)$ time. To allocate the processors, we assign $2(r-1)$ processors to each item in NEWUP(u). Each item participates in $r-1$ merging problems. In a merging problem, if the item is part of a set of size r, the item uses one of its assigned processors. If the item is part of a set of size $<r$, the item takes one processor from the item e defining the interval I. Each item e contributes at most $r-1$ processors to a merging problem in the latter manner, thus it suffices to provide $2(r-1)$ processors to each item in NEWUP(u).

We conclude

Theorem 3: There is a CRCW sorting algorithm for the CRCW PRAM that uses $n \leq p \leq n^2$ processors and runs in time $O(\frac{\log n}{1+\log\log 2p/n})$.

5. A Parametric Search Technique

The reader is warned that this section is not self contained. We recall Megiddo's parametric search technique [M-83] and its improvement in many instances in [C-87]. The improvement was an asymptotic improvement, but was not practical for it was based on the AKS sorting network. As we will explain, the role played by the AKS network can be replaced by the EREW sorting algorithm from Section 3.

In a nutshell, the procedure of [C-87] can be described as follows. A comparison based sorting algorithm is executed; however each 'comparison' is an expensive operation costing $C(n)$ time, where n is the size of the problem at hand. In addition, the comparisons have the property that they can be 'batched': given a set of c comparisons, one of them can be evaluated, and the result of this evaluation resolves a further $c/2$ comparisons, in an additional $O(c)$ time. Examples of search problems (called *parametric search problems*), mostly geometric search problems, for which this approach is fruitful, include [M-83, C-87, C-85, CSS-86]. Megiddo showed that parallel sorting algorithms, executed sequentially, provide good sorting algorithms for this type of problem; the reason is that a parallel sorting algorithm naturally batches comparisons.

In [C-87] it was shown how to achieve a running time of $O(n\log n + \log n\, C(n))$ for the parametric search problems, when using a depth $O(\log n)$ sorting network as the sorting algorithm. (Briefly, the solution required $O(\log n)$ 'comparisons' to be evaluated; the overhead for running the sorting algorithm and selecting the comparisons to be evaluated was

O(n logn) time.) It was also observed that to achieve this result it sufficed to have a comparison based algorithm which could be represented as an O(logn) depth, O(n) width, directed acyclic graph, having bounded in degree, where each node of the graph represented a comparator and the edges carried the outputs of the comparators.

In fact, a slightly more general claim holds. We start by defining a *computation graph* corresponding to an EREW PRAM algorithm on a given input. We define a parallel EREW PRAM computation to proceed in a sequence of steps of the following form. In each step, every processor performs at most b (a constant) reads, followed by at most b writes; a constant number of arithmetic operations and comparisons are intermixed (in any order) with the reads and writes. We represent the computation of the algorithm on a given input as a computation graph; the graph has depth 2T (where T is the running time of the algorithm) and width M+P (where M is the space used by the algorithm and P is the number of processors used by the algorithm). In the computation graph each node represents either a memory cell at a given step, or a processor at a given step. There is a directed edge $(<m,t>,<p,t>)$ if processor p reads memory cell m at step t; likewise there is a directed edge $(<p,t>,<m,t+1>)$ if processor p writes to memory cell m at step t. If no write is made to memory cell m at step t, there is a directed edge $(<m,t>,<m,t+1>)$.

Suppose we restrict our attention to algorithms such that at the start of the algorithm, for each memory cell (at step t+1) we know whether the in-edge (in the computation graph) comes from a processor (at step t) or a memory cell (at step t). For a sorting algorithm of this type, that runs in time O(logn) on n processors using O(n) space, we can still achieve a running time of O(n logn + lognC(n)) for the parametric search problems. (To understand this, it is necessary to read [C-87], Sections 1-3. The reason the result holds is that we can determine when a memory cell is active, to use the terminology of [C-87], and thus play the game of Section 2 of [C-87] on the computation graph. In general, if we do not have the condition on the in-edges for memory cells, it is not clear how to determine if a memory cell is active. As explained in Section 3 of [C-87], given a solution to the game of Section 2, we can readily obtain a solution to the parametric search problem).

Remark: The computation graph need not be the same for all inputs of size n. In addition, the graph does not have to be explicitly known at the start of the sorting algorithm.

Next, we show that the EREW sorting algorithm satisfies the conditions of the previous paragraph. Each of the five steps for one stage of the EREW algorithm comprises the computation of the cross-ranks of two arrays. The computation of the cross-ranks proceeds in two substeps; in the first substep, each processor performs a constant number of reads; in the second substep, each processor performs a constant number of writes.

We conclude that the result of [C-87] can be achieved with a much smaller constant, thereby making the paradigm less impractical.

Acknowledgements.

Thanks to Richard Anderson and Allen Van Gelder for commenting on an earlier version of this result. In particular, Allen Van Gelder suggested the term *cross-rank*. Many thanks to Jeanette Schmidt for questions and comments on several earlier versions of the result. Thanks also to Alan Siegel and Michelangelo Grigni for suggestions that simplified the presentation. Finally, I am grateful to Zvi Kedem, Ofer Zajicek, and Wayne Berke, who all read the penultimate version of the paper, and provided a variety of useful comments.

References

[AKS-83] M. Ajtai, J. Komlos, E. Szemeredi, "An O(nlogn) sorting network", *Combinatorica*, 3(1983), 1-19.

[AAV-86] N. Alon, Y. Azar, U. Vishkin, "Tight complexity bounds for parallel comparison sorting", *Twenty Seventh Annual Symposium on Foundations of Computer Science*, 502-510.

[B-68] K. E. Batcher, "Sorting Networks and their applications", *Proc. AFIPS Spring Joint Summer Computer Conf.*, Vol.32, 307-314.

[BH-82] A. Borodin and J. Hopcroft, "Routing, merging and sorting on parallel models of computation", *Proc. Fourteenth Annual ACM Symp. on Theory of Computing, 338-344.*

[BH-87] P. Beame and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM", manuscipt.

[BN-86] G. Bilardi and A. Nicolau, "Bitonic sorting with O(nlogn) comparisons", *Proc. Twentieth Annual Conf. on Information Sciences and Systems.*

[C-85] R. Cole, "Partitioning point sets in arbitrary dimension", Computer Science Department Technical Report #184, Courant Institute, accepted, *Special TCS issue / ICALP '85.*

[C-87] R. Cole, "Slowing down sorting networks to obtain faster sorting algorithms", *JACM*, Vol.34, No.1, 200-208.

[CO-86] R. Cole, C. O'Dunlaing, "Notes on the AKS sorting network", Computer Science Dept. Technical Report #243, Courant Institute.

[CSS-86] R. Cole, J. Salowe and W.L. Steiger, "Selecting slopes", Submitted for publication.

[CV-87] R. Cole, U. Viskin, "Faster optimal prefix sums and list ranking", Computer Science Department Technical Report #277, Courant Institute.

[LPS-86] A. Lubotzky, R. Phillips, and P. Sarnak, "Ramanujan conjecture and explicit constructions of expanders and super-concentrators", *Eighteenth Annual Symposium on Theory of Computing*, 240-246.

[K-83] C. Kruskal, "Searching, merging, and sorting in parallel computation", *IEEE Trans. Comp.*, Vol.C-32, No.10, 942-946.

[M-83] N. Megiddo, "Applying parallel computation algorithms in the design of serial algorithms", *JACM, 30, 4(1983), 852-865.*

[Pa-87] M. S. Paterson, "Improved sorting networks with O(logn) depth", Dept. of Computer Science Research Report RR89, University of Warwick.

[Pr-78] F. P. Preparata, "New Parallel-Sorting Schemes", *IEEE Transactions on Computers*, Vol. C-27, No. 7, 669-673.

[V-75] L. Valiant, "Parallelism in comparison problems", *SIAM J. Comput*, Vol. 4, 348-355.

NYU COMPSCI TR-278 c.1
Col

NY
Co
Pa

JA

NYU COMPSCI TR-278 c.1
Cole, Richard
Parallel merge sort.

JAN 1 1988	[illegible]

This book may be kept JUN. 0 1 1987

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

JAN 1 1988			

GAYLORD 142 PRINTED IN U.S.A.